معالجة الاعتراضات ***Exception Handling***

هل تصدق أن أسهل مرحلة من مراحل بناء البرنامج هي كتابة الكود؟
نعم أنا متأكد من كلامي، إذ أن المرحلة الأصعب منها كثيراً هي مرحلة وضع model مناسب للبرنامج ودراسة بنيته وعلاقة أقسامه ببعضها، بحيث تجنب هذه المرحلة[1] المبرمج الكثير من الأخطاء والمشاكل في البرنامج.
ولكن مع ذلك لا يكاد يخلو أي كود من الأخطاء، لذا كان لابد من الاهتمام بمعالجة الأخطاء كمرحلة هامة جداً[2] من مراحل بناء أي برنامج..

ذكرنا مسبقاً أن الأخطاء في Java تقسم إلى قسمين:

١. *Compile-Time errors*: وهي الأخطاء التي يستطيع compiler الـ Java اكتشافها أو التنبؤ بحدوثها قبل تنفيذ البرنامج أي في وقت الترجمة، وتصنف الكثير من الأخطاء تحت هذا العنوان كالأخطاء القواعدية أو عدم عمل Initialization للمتحولات..
كما أن compiler الـ Java يعتبر ذكياً، إذ أنه يستطيع التنبؤ بالكثير من الأخطاء التي قد تحدث أثناء التنفيذ وينبه المبرمج عليها في وقت الترجمة، مثل خطأ القسمة على صفر في المثال التالي:

مثال:

```
int i = 0;
int j = 5/i;
```

في هذه الحالة سيكتشف الـ compiler أن لدينا قسمة واضحة على الصفر لذا سيعترض..

٢. *Run-Time error*: هي أخطاء لا يمكن اكتشافها أثناء الترجمة، وتقسم إلى قسمين:
   a. Errors: وهي أخطاء لا يمكن معالجتها برمجياً لذا لن نبحث فيها، ومثالها: امتلاء الذاكرة..
   b. Exceptions: وهي اعتراضات تظهر أثناء التنفيذ وستكون موضوع دراستنا في هذه المحاضرة.

الاعتراضات Exceptions:

مع بناء التطبيقات الكبيرة ومنها أنظمة التشغيل، ظهرت الحاجة الشديدة لوضع آلية لمعالجة الاعتراضات بشكل فعال بدون الحاجة إلى *نكت البرنامج* جرياً وراء خطأ ما، وخصوصاً أن كل مبرمج يعتقد أن كوده خالٍ تماماً من الأخطاء (١٠٠% Bugs Free)!!!! لذا بدأت لغات البرمجة بوضع هذه الآلية وتطويرها.

---
1 إن كانت مدروسة جيداً
2 حتماً هي أصعب من كتابة الكود

تميزت Java عن جميع لغات البرمجة بآلية متينة وقوية جداً لمعالجة الاعتراضات، حيث أحاطت هذه الآليـــة بالبرنامج بحيث منعت أي انهيار مفاجئ وغير متوقع للبرنامج.

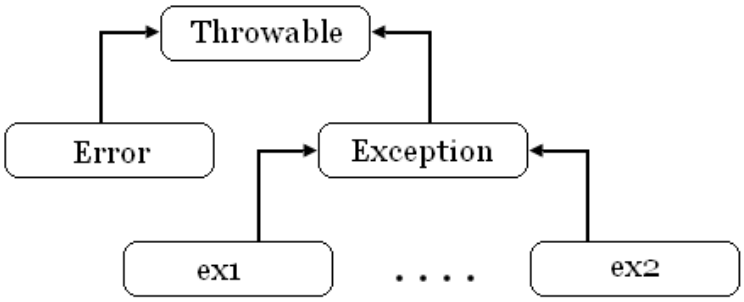
فمثلاً: في بعض لغات البرمجة مثل الـــ C++، لا تعترض اللغة أبداً عند طلب عنصر من المصفوفة خارج عن مجال تعريفها[1] وقد ينتج عن هذا أننا قد نتجاوز حدود الذاكرة المخصصة لبرنامجنا وقد نغيـــر فـــي متحـــولات برنامج آخر وبالتالي تحدث المصائب!!

هذا الكلام مرفوض تماماً في Java إذ أنها تنبه المبرمج إلى أنه تجاوز حدود مصفوفته عن طريق اعتـــراض (Exception).

ما هو الـــ Exception؟

في الحقيقة الاعتراض في Java هو object من صف خاص يعبر عن نوعية الخطأ الذي أدى إلى حدوث هذا الاعتراض، إذ أن لكل نوع من أنواع الاعتراضات class خاص به ينشأ غرض منه عند حدوث هذا النوع من الأخطاء ويرمى (throw) في وجه تنفيذ البرنامج..

إن جميع هذه الصفوف موروثة من الصف Exception وهو بدوره مشتق من الصف Throwable والـــذي يشكل أيضاً أباً للصف Error وهو بدوره أب لجميع الأخطاء التي لا تعالج برمجياً..



كيف يعمل الـــ Exception؟

عندما ينشأ أي خطأ يقوم الـــ JVM بما يلي:

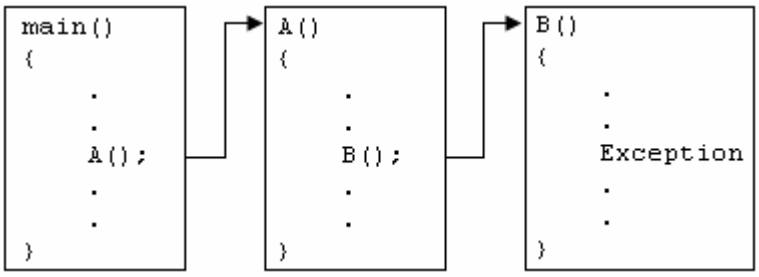
- يتوقف تنفيذ الإجرائية التي ظهر الاعتراض فيها، ولا يتابع تنفيذها أبداً[2].
- يتولد الاعتراض المناسب لهذا الخطأ، أي ينشأ object من الصف الذي يمثل هذا الخطأ.
- **يرمى (throw)** هذا الـــ object.

[1] مثلاً لدينا مصفوفة a طولها ٤ وقمنا بطلب a[6]

[2] لن يتم تنفيذ باقي الكود حتى بعد معالجة الاعتراض كما سنرى

ماذا نقصد برمي الـــ object؟

للنظر إلى المخطط التالي:



نلاحظ ظهور اعتراض في الإجرائية ()B، فإذا قام المبرمج **بالتقـــاط (catch)** هذا الاعتراض *أي معالجته* مباشرة حلت المشكلة وتابع البرنامج عمله بشكل طبيعي وسندرس آلية المعالجة بعد قليل.

ولكن ما الذي سيحدث فيما لو لم يعالج المبرمج هذا الخطأ؟

هنا تبرز روعة Java، حيث أن الـــ JVM إذا لم يعثر على الخطأ في الإجرائية ()B سيقوم **برمي** الاعتراض إلى المستوى الأعلى أي إلى الإجرائية التي قامت باستدعاء ()B وهي في مثالنا ()A.

بنفس الطريقة إذا تم التقاط الاعتراض في ()A حلت المشكلة، وإلا سيرمى إلى المستوى الأعلى وهكذا..

خلاصة:

إن أي إجراء ينشأ فيه اعتراض يتوقف تنفيذ تعليماته، وينشأ object يمثل الاعتراض الذي نـــشأ نتيجـــة هـــذا الخطأ، فإن التقط هذا الاعتراض توقف الخطأ وعاد البرنامج للتنفيذ بشكل طبيعي[1]، أي إن المستويات الأعلى لن تشعر بأي شيء، وإلا فإن الاعتراض سيرمى إلى المستوى الأعلى حيث تتم نفس المناقشة.

ماذا لو وصلنا إلى الـــ ()main ولم يلتقط الاعتراض؟

في هذه الحالة سيلقى الخطأ إلى الـــ JVM ويقوم هذا الأخير بإيقاف عمل البرنامج كله مع رسالة تنبيه.

معالجة الاعتراضات:

تعرفنا حتى الآن على الاعتراضات وكيفية تصرف الـــ JVM عند نشوء أحدها، ولكن كيف سنعالجها؟

في لغات البرمجة القديمة كان على المبرمج أن يملئ برنامجه بالاختبارات كي يعالج جميع حـــالات الأخطـــاء، وغالباً ما يغيب عن باله معالجة نوع ما من أنواع الأخطاء لكثرتها وصعوبة الإحاطة بها جميعـــاً أثنـــاء كتابـــة البرنامج..

أما في Java وكما توضح قبل قليل فإن معالجة الاعتراض تكون بالتقاطه (catch) ولكن ماذا نعني بذلك؟

بما أن الاعتراض هو عبارة عن object يرمى من المستوى الأدنى إلى المستويات الأعلى، فإن علينا تطويقـــه ومنعه من متابعة طريقه، ويتم ذلك عن طريق التعليمة try-catch.

[1] سيتضح بعد قليل أي من التعليمات سيعود التنفيذ إليها

سنضرب مثالين الأول يستخدم الطريقة القديمة في معالجة الأخطـــاء والثـــاني يـــستخدم التعليمـــة try-catch وسنوضح من خلاله طريقة عمل هذه التعليمة:

مثال(١):

```
void A(int[] a, int i) {
    if (a != null)
        if (i >= 0 && i < a.length)
            if (a[i] != 0)
                System.out.println(100 / a[i]);
            else
                System.out.println("ArithmeticException: / by zero..");
        else
            System.out.println("ArrayIndexOutOfBoundsException..");
    else
        System.out.println("NullPointerException..");
}
```

التعليمة الأساسية في هذا المثال هي:

```
System.out.println(100 / a[i]);
```

نلاحظ أننا ناقشنا في هذا المثال ٣ احتمالات لأخطاء ممكن أن تحدث في هذا التابع وهي:

١. أن يكون مؤشر المصفوفة يؤشر إلى *null*.
٢. أن يكون i خارج حدود المصفوفة.
٣. أن تكون قيمة الحقل a[i] مساوية للصفر وبالتالي يكون لدينا مشكلة قسمة على الصفر.

مثال(٢):

```
void B(int[] a, int i) {
    try {
        System.out.println(100 / a[i]);
    }
    catch (NullPointerException e) {
        System.err.println(e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println(e);
    }
    catch (ArithmeticException e) {
        System.err.println(e);
    }
    catch (Exception e) {
      ...
    }
}
```


ما الجديد في هذا المثال؟

سنشرح آلية عمل التعليمة (try-catch) ثم نقارن مع المثال السابق:

عندما نتوقع أن تظهر أخطاء في كتلة برمجية ما، فما علينا إلا أن نضعها ضمن scope التعليمة try ومن ثـــم تقوم التعليمة catch بالتقاط الاعتراض ومنعه من الصعود إلى المستوى الأعلى.

خصائص التعليمة catch:

- تأخذ هذه التعليمة متحولاً واحداً فقط هو object من الـ class الذي يمثل نوع الخطأ الذي ترغب هذه التعليمة بمعالجته، وهذا يعني أن علينا وضع تعليمة catch خاصة بكل نوع من أنواع الاعتراضات.
- عندما يظهر خطأ ما ضمن scope التعليمة try، فإن تنفيذ تعليماتها يتوقف، وينتقل التنفيذ مباشرة إلى خارج scope التعليمة try حيث يتم اختبار تعليمات catch بالترتيب من الأعلى إلى الأسفل، فإذا تم التقاط الخطأ في أحد تعليمات catch فإن التنفيذ ينتقل إليها وتهمل باقي التعليمات.
- بعد الانتهاء من تنفيذ الكتلة catch سينتقل التنفيذ إلى ما بعد جميع التعليمات catch ويتابع البرنامج سيره بشكل طبيعي.
- إذا لم تستطع أي تعليمة catch التقاط الخطأ فسيرمى إلى المستوى الأعلى وكأن تعليمة try-catch غير موجودة أصلاً.
- يمكن تطبيق مفهوم الـ polymorphism في هذه التعليمة، إذ أننا عندما نمرر لها مؤشراً من نمط اعتراض ما، فإن أي اعتراض من هذا النمط أو أحد أبنائه سيلتقط.
- نستفيد من الفكرة السابقة في وضع تعليمة catch تستطيع التقاط أي اعتراض قد ينشأ ضمن الكتلة البرمجية، وذلك بتمرير مؤشر من النمط Exception إذ أننا نعلم أن جميع الاعتراضات ترث من هذا الصف.
- يفترض أن تكون المعالجة السابقة آخر معالجة ضمن تعليمات catch لأنها ستنفذ عند أي اعتراض وبالتالي ستحجب كل ما تحتها، وقد يكون تحتها معالجة خاصة أكثر بنوع الخطأ الذي يرتكب.
- الغرض المؤشر عليه بـ e والذي يمرر للتعليمة يحوي عدة توابع مفيدة، منها مثلاً تابع الـ ()toString الذي استخدمناه في الطباعة في المثال السابق، والذي يطبع عبارة توضح نوع الخطأ.

في كلى المثالين السابقين قمنا بمعالجة الأخطاء فما الفرق بينهما؟

لنقارن بين المثالين السابقين:

❖ نلاحظ أن المعالجة في المثال الأول أدت إلى تغيير في الكود الأصلي الذي كان يقتصر على تعليمة واحدة فقط، فتخيل لو أن لدينا تابع مكون من ١٠٠ تعليمة وفجأة ظهر خطأ لا يعرف سببه فكيف ستكون التعديلات على الكود؟!؟!؟

❖ بينما نلاحظ أن المثال الثاني قام بفصل معالجة الأخطاء فصلاً تاماً عن الكود الأصلي، كما أنه عالج حالة ظهور أي خطأ لا حقاً عن طريق التعليمة:

```
catch (Exception e) {
  ...
}
```

أي أن ظهور أي خطأ في البرنامج لن يكلفني كركبة الكود بأكمله..

التعليمة finally:

ذكرنا أن التنفيذ لن يقترب من التعليمات catch إلا في حالة ظهور exception في الكود الموجود ضمن scope التعليمة try، ولكن ماذا لو أردنا وضع مجموعة من التعليمات تنفذ دوماً عند ظهور الخطأ أو عدم ظهوره؟

هنا يأتي دور التعليمة finally التي ينفذ الـ scope الخاص بها دوماً بعد الانتهاء من تنفيذ التعليمات ضمن try، فإذا ظهر اعتراض ضمن الـ try ينتقل التنفيذ إلى الـ catch المناسبة وبعدها فوراً سينتقل التنفيذ إلى التعليمة finally، وإلا فسينتقل التنفيذ مباشرة إلى finally.

```
try{}
catch (ex1 e){}
catch (ex2 e){}
catch (ex3 e){}
finally {}
```

فائدة هذه التعليمة كبيرة جداً إذ أنها تقوم إلى حد ما بدور شبيه بدور الهادم (destructor)، حيث يقوم المبرمج بالأعمال التي يريدها وهو مطمئن إلى أن محتوى هذه التعليمة سينفذ في كل الحالات.

ولكننا نعلم في Java أن الـ garbage collector يقوم بهدم جميع الـ objects فما الفائدة منها إذاً؟

فائدتها تكمن في الأشياء التي لا يقوم بها الـ gc، مثل مسح شكل من على الشاشة أو إغلاق ملف مفتوح أو..

مثلاً:

ليكن لدينا صف يقوم بالتعامل مع الملفات، فإذا قمنا بعملية فتح ملف ومن ثم حدث exception قبل أن نقوم بإغلاقه سيبقى مفتوحاً، والحل هو استخدام التعليمة finally ووضع تعليمة الإغلاق ضمنها.

الصفة throws:

يجب النظر للبرنامج كوحدة متكاملة، حيث أننا لا نستطيع دوماً التقرير في ما سنفعله تجاه الاعتراض الناشئ من المستوى المباشر الذي ظهر فيه، وإنما قد يكون الإجراء ذو المستوى الأعلى أقدر على حل المشكلة من الإجراء الذي ظهر فيه الاعتراض.

مثلاً: لنعد إلى مثال الصف الذي يتعامل مع الملفات (فتح، إغلاق..) ولنفرض أن تابع فتح الملف لم يعثر على أي ملف مطابق للمسار الذي أعطي له، فما الحل؟

سيحتار مبرمج هذا الصف، إذ أنه من الممكن أن يظهر رسالة خطأ أو..

هناك الكثير من الاحتمالات، ولكن الأقدر على اتخاذ هذا القرار هو المبرمج الذي يستخدم غرضاً من الصف السابق، إذ أنه هو من أدخل مسار الملف إلى ذاك الغرض، لذا هو أعلم بنوع المشكلة وكيفية حلها.

لذا فالحل هنا هو أن يجبر التابع (openFile()) الذي يقوم بفتح الملف كل من يستخدمه على معالجة احتمال ظهور أخطاء في فتح الملف..

وكمثال آخر: لنعد إلى المثال(٢) الذي مر قبل قليل، نلاحظ أن الإجرائية B ليست أفضل من يحل المشاكل التي ناقشنا حالة ظهورها، لذا يمكن أن تجبر هذه الإجرائية كل من يستدعيها على معالجة جميع تلك الأنواع من الأخطاء.

يتم هذا الإجبار عن طريق التعليمة throws، حيث سنخبر المترجم بأن الإجرائية B يمكن أن ترد الأنواع التالية من الـ exceptions، وعندها يكون أمام كل من يريد أن يستخدم هذه الإجرائية حل من اثنين:

١. إما أن يقوم بمعالجة الاعتراض عن طريق التعليمة try-catch، حيث يجبره المترجم على معالجة جميع أنواع الأخطاء التي يمكن أن يردها الإجراء B، وذلك إما بمعالجة كل اعتراض على حدة أو بمعالجة اعتراض واحد يكون أباً لجميع تلك الاعتراضات[1].
٢. إذا أحس هذا الإجراء أن المستوى الأعلى منه هو الأقدر على حل المشاكل يمكن أن يرمي بدوره الأخطاء إلى المستوى الأعلى منه عن طريق throws أيضاً.

إذا لم يطبق أحد الخيارين السابقين فالنتيجة ستكون خطأ Compiler.

مثال:

```
class TestThrows
{
    void B(int[] a, int i) throws NullPointerException,
                                  ArrayIndexOutOfBoundsException,
                                  ArithmeticException
    {
        System.out.println(100 / a[i]);
    }

    void test () {
        try {
            int[] a = {1,2,3,4};
            this.B(a, 2);
        }
        catch (NullPointerException e) {
            System.err.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.err.println(e);
        }
        catch (ArithmeticException e) {
            System.err.println(e);
        }
        catch (Exception e) {
            ...
        }
    }
}
```


ملاحظات:

١. إن الاعتراض إن لم يعالج في أي مستوى فسيرمى للمستوى الأعلى في حالة وجود هذه التعليمة أو عدم وجودها، ولكن وجودها يجبر المستوى الأعلى إجباراً على معالجة الاعتراض أو رميه إلى ما فوقه..

[1] مثل الاعتراض Exception الذي يكون أباً لجميع الاعتراضات كما ذكرنا مسبقاً

٢. يمكن أن يرمي كل إجراء الاعتراض إلى الإجراء ذي المستوى الأعلى حتى نصل إلى الـ main الذي من الممكن أن يقوم بدوره برمي الاعتراض إلى الـ JVM وعندها يتوقف البرنامج[١]!!

التعليمة throw:

يمكن أن يعالج الاعتراض في أكثر من مستوى، كل على طريقته وحسب زاوية رؤيته لهذا الاعتراض..
ولكننا نعلم أن معالجة الاعتراض في أي مستوى تمنعه من الصعود إلى المستوى الأعلى، فكيف سنطبق هذا الكلام؟
الحل يكون عن طريق إعادة رمي الاعتراض مرة أخرى بعد الإمساك به، وذلك ضمن التعليمة catch ولكن بشرط أن تكون الإجرائية تحقق صفة (*throws*) لهذا النوع من الأخطاء.

مثال:

```
void testThrow () throws Exception {
	try {

	}
	catch (Exception e){
		throw e;
	}
}
```

بناء الـ Exceptions يدوياً:

عندما نبرمج class خاص بنا من المنطقي أن نبني له مجموعة من الـ Exceptions التي تضبط استخدام المبرمجين الآخرين له.
وبما أن الاعتراض هو class فإن إنشاء أي Exception جديد لا يعدو إنشاء class مشتق من الصف Exception وإعادة تعريف ما يلزمنا من توابع.
ويجب أن يكون التابع الذي سيرمي الاعتراض يحقق صفة (*throws*) لهذا النوع من الاعتراضات.

مثال:

```
class ZeroDenominatorException extends Exception
{
	public ZeroDenominatorException() {}

	public ZeroDenominatorException(String msg) { super(msg); }

	public String getMessage() {
		return "Detail Message: " + super.getMessage();
	}

	public String toString() {
		return "ZeroDenominatorException Exception ..";
	}
}
```

[١] لا أعتقد أن مبرمجاً محترفاً سيجرب هذا الحل (الظريف) في برنامجه!!

```
class Fraction
{
      private int numerator, denominator;
      public Fraction(int numerator, int denominator)
                                    throws ZeroDenominatorException {
            if (denominator == 0)
                  throw new ZeroDenominatorException();
            this.numerator = numerator;
            this.denominator = denominator;
      }
}
```

:(*throws*) and Overriding

مثال:

```
class A {
      void b() throws NullPointerException {}
}
```

عند الوراثة من الصف A وإعادة تعريف (Overriding) الإجرائية b()، فإن المترجم يمنع الإجرائية b() في الابن من رمي أي نوع من الاعتراضات عدا الاعتراضات المعرفة في الإجرائية الأصلية (الموجودة في الأب) عن طريق الصفة *throws*.

ولهذا العنوان تفصيل في المرجع ولكنه مزعج قليلاً وقد ضرب عليه مثال طويل، فمن أحب أن يستزيد فيه فعليه مراجعة المرجع في نهاية الفصل التاسع تحت عنوان: *Exception restrictions*.

خلاصة:

١. تتم معالجة الاعتراضات عن طريق التعليمة try والتعليمات المرافقة catch و finally.
٢. عند حدوث اعتراض ضمن كتلة try يتوقف تنفيذ باقي تعليماتها ولا يعود إليها مطلقاً.
٣. يتم فحص تعليمات catch الأقرب فالأبعد حتى نعثر على تعليمة دخلها من نفس نوع الغرض الذي أنشأه الاعتراض أو من أحد آبائه، فإن استطاعت أحدها إمساكه حجبت الباقي، وإلا رمي إلى المستوى الأعلى.
٤. بعدها ينتقل التنفيذ إلى finally دوماً (إن وجدت) وذلك إن كان هناك اعتراض أو لا، وإن وجد الاعتراض فستنفذ finally سواء التقط أو لا.
٥. مهمة الصفة (throws) هي إجبار المستويات الأعلى على معالجة الاعتراض، حيث لن يسمح المترجم باستدعاء الإجرائية ذات الصف throws إلا ضمن تعليمة try-catch أو أن تكون الإجرائية المستدعية هي نفسها throws لنفس أنواع الاعتراضات.
٦. يقوم التابع throw برمي الاعتراضات سواء منها الموجود أصلاً في Java أو الذي عرفناه يدوياً أو إعادة رمي اعتراض تم إمساكه كما وضحنا سابقاً.

تنبيهات:

١. لاشيء يمنع احتواء الـــ scope الخاص بالتعليمتين catch و finally على معالجة للاعتراضات وبالتالي على تعليمة try-catch.
٢. حاول دوماً إمساك الاعتراض عن طريق نوعه مباشرة[1] لأن نوع المعالجة يختلف من اعتراض لآخر.
٣. انتبه إلى الاعتراضات التي تظهر في الباني (constructor) لأنها تكون حساسة أكثر من غيرها، وذلك لأننا لا ندري هنا أي العناصر أنشئت وأيها لما ينشأ بعد.
٤. يقوم البعض بترقيع برنامجهم بطريقة غير لائقة عن طريق وضع التعليمة:

```
catch(Exception e) {}
```

وترك الـــ scope خالياً وعندها ستلتقط جميع الأخطاء ويستمر عمل البرنامج دوماً، ولكن هذا التـــصرف معيب جداً إذ أن المبرمج لا يدري بالأخطاء والمصائب التي يمررها البرنامج، وبالتالي ســـيفقد مـــصداقية العمليات التي يقوم بها إذ أن احتمال إعطائها أجوبة خاطئة أصبح كبيراً.
٥. إذا تم الالتزام بالطريقة السليمة لاستخدام هذه القوة و المتانة في Java فسينتج برنامج قوي ومتماسك ومن الصعب أن ينهار بشكل مفاجئ كما في لغات أخرى.

انتهت المحاضرة

[1] وليس كما يفعل البعض حيث يلتقط جميع الأخطاء عن طريق نفس التعليمة catch وهي: catch(Exception e) لأنه سيخسر إمكانية معالجة كل اعتراض بالطريقة السليمة